Large-scale neighbor-joining with NINJA

Travis J. Wheeler

twheeler@cs.arizona.edu
Department of Computer Science
The University of Arizona, Tucson AZ 85721, USA

Abstract Neighbor-joining is a well-established hierarchical clustering algorithm for inferring phylogenies. It begins with observed distances between pairs of sequences, and clustering order depends on a metric related to those distances. The canonical algorithm requires $O(n^3)$ time and $O(n^2)$ space for n sequences, which precludes application to very large sequence families, e.g. those containing 100,000 sequences. Datasets of this size are available today, and such phylogenies will play an increasingly important role in comparative genomics studies. Recent algorithmic advances have greatly sped up neighbor-joining for inputs of thousands of sequences, but are limited to fewer than 13,000 sequences on a system with 4GB RAM. In this paper, I describe an algorithm that speeds up neighbor-joining by dramatically reducing the number of distance values that are viewed in each iteration of the clustering procedure, while still computing a correct neighbor-joining tree. This algorithm can scale to inputs larger than 100,000 sequences because of external-memory-efficient data structures. A free implementation may by obtained from http://nimbletwist.com/software/ninja.

Keywords Phylogeny inference, Neighbor joining, external memory

1 Introduction

The neighbor-joining (`NJ`) method of Saitou and Nei [1] is a widely-used method for constructing phylogenetic trees, owing its popularity to good speed, generally good accuracy [2], and proven statistical consistency (informally: `NJ` reconstructs the correct tree given a sufficiently long sequence alignment) [3–5].

`NJ` is a hierarchical clustering algorithm. It begins with a distance matrix, where d_{ij} is the observed distance between clusters i and j, and initially each of the n input sequences forms its own cluster. `NJ` repeatedly joins a pair of clusters that are closest under a measure, q_{ij}, that is related to the d_{ij} values. The canonical algorithm [6] finds the minimum q_{ij} at each iteration by scanning through the entire current distance matrix, requiring $O(r^2)$ work per iteration, where r is the number of remaining clusters. The result is a $\Theta(n^3)$ run time, using $\Theta(n^2)$ space. Thus, while `NJ` is quite fast for n in the hundreds or thousands, both time and space balloon for inputs of tens of thousands of sequences.

As a frame of reference, there are 8 families in Pfam [7] containing more than 50,000 sequences, and 3 families in Rfam [8] with more than 100,000 sequences, and since the number of sequences in genbank is growing exponentially [9], these numbers will certainly increase. Phylogenies of such size are applicable, for example, to large-scale questions in comparative genomics (e.g. [10]).

Related work `QuickTree` [11] is a very efficient implementation of the canonical `NJ` algorithm. Due to low data-structure overhead, it is able to compute trees up to nearly 40,000 sequences before running out of memory on a 4GB system. `QuickJoin` [12, 13], `RapidNJ` [14], and the

bucket-based method of [15] all produce correct NJ trees, reducing run time by finding the globally smallest q_{ij} without looking at the entire matrix in each iteration. While all methods still suffer from worst-case running time of $O(n^3)$, they offer substantial speed improvements in practice. Unfortunately, the memory overhead of the employed data structures reduces the number of sequences for which a tree can be computed (e.g. on a system with 4GB RAM, RapidNJ scales to 13,000 sequences, and QuickJoin scales to 8000).

The focus of this paper is on exact NJ tools, but I briefly mention other distance-based methods for completeness. Relaxed [16] and fast [17] neighbor joining are NJ heuristics that improve speed by choosing the pair to merge from an incomplete subset of all pairs; they do not guarantee an exact NJ tree in the typical case that pairwise distances are not nearly-additive (very close to the distances induced by the true tree). Minimum-evolution with NNI offers an alternate fast approach with good quality and conjectured consistency [19]. An implementation of the relaxed neighbor joining heuristic, ClearCut [18], is faster than NINJA on very large inputs, and a recent implementation of a minimum-evolution heuristic with NNI, FastTree [20], is notable for constructing accurate trees on datasets of the scale discussed here, with speed at least 10-fold greater than that acheived by NINJA on very large inputs.

Contributions I present NINJA, an implementation of an algorithm in the spirit of QuickJoin and RapidNJ: it produces a correct NJ phylogeny, and achieves increased speed by restricting its search for the smallest q_{ij} at each iteration to a small portion of the quadratic-sized distance matrix. The key innovations of NINJA are (1) introduction of a search-space filtering scheme that is shown to be consistently effective even in the face of difficult inputs, and (2) inclusion of data structures that efficiently use disk storage as external memory in order to overcome input size limits.

The result is a statistically consistent phylogeny inference tool that is roughly an order of magnitude faster than a very fast implementation of the canonical algorthm, QuickTree (for example, calculating a NJ tree for 60,000 sequences in less than a day on a desktop computer), and is scalable to hundreds of thousands of sequences.

Overview The next section gives necessary details of the canonical NJ algorithm. Section 3 describes the primary filtering heuristic used to avoid viewing most of the distance matrix at each iteration, called *d-filtering*. Section 4 describes a secondary filtering method, called *q-filtering*, which is of primary value on the kinds of inputs where d-filtering is ineffective. Section 5 gives the full algorithm, and finally section 6 analyzes the impact of these methods, and compares the scalability of NINJA to that of other exact neighbor joining tools.

2 Canonical neighbor-joining

NJ [1, 6] is a hierarchical clustering algorithm. It begins with a distance matrix, D, where d_{ij} is the observed distance between clusters i and j, and initially each sequence forms its own cluster. NJ forms an unrooted tree by repeatedly joining pairs of clusters until a single cluster remains. At each iteration, the pair of clusters merged are those that are closest under a transformed distance measure

$$q_{ij} \;=\; (r-2)\, d_{ij} \;-\; t_i \;-\; t_j \;, \tag{1}$$

where r is the number of clusters remaining at the time of the merge, and

$$t_i \;=\; \sum_k d_{ik} \;. \tag{2}$$

When the $\{i, j\}$ pair with minimum q_{ij} is found, D is updated by inactivating both the rows and columns corresponding to clusters i and j, then adding a new row and column containing the distances to all remaining clusters for the newly formed cluster ij. The new distance $d_{ij|k}$ between the cluster ij and each other cluster k is

$$d_{ij|k} \;=\; (d_{i|k} \;+\; d_{j|k} \;-\; d_{i|j})/2\;. \tag{3}$$

There are n-1 merges, and in the canonical algorithm each iteration takes time $O(r^2)$ to scan all of D. This results in an overall running time of $O(n^3)$.

3 Restricting search of the distance matrix

3.1 The d-filter

A valid filter must retain the standard `NJ` optimization criterion at each iteration: merge a pair $\{i, j\}$ with smallest q_{ij}. To avoid scanning the entire distance matrix D, the pairs can be organized in a way that makes it possible to view only a few values before reaching a bound that ensures that the smallest q_{ij} has been found.

To acheive this we use a bound that represents a slight improvement to that used in `RapidNJ` [14]. In that work, $(\{i, j\}, d_{ij})$ triples are grouped into sets, sorted in order of increasing d_{ij}, with one set for each cluster. Thus, when there are r remaining clusters, each cluster i has a related set S_i containing $r - 1$ triples, storing the distances of i to all other clusters j, sorted by d_{ij}. Then, for each cluster i, S_i is scanned in order of increasing d_{ij}. The value of q_{ij} is calculated (equation 1) for each visited entry, and kept as $q_{\min}$ if it is the smallest yet seen.

To limit the number of triples viewed in each set, a second value is calculated for each visited triple, a lower bound on q-values among the unvisited triples in the current set S_i: $q_{\text{bound}} \;=\; (r-2)\; d_{ij} \;-\; t_i \;-\; t_{\max}$, where $t_{\max} \;=\; \max_{\text{k}}\{t_k\}$. In a single iteration, $t_{\max}$ is constant, and for a fixed set S_i, t_i is constant and the sorted d_{ij} values are by construction non-decreasing. Thus, if $q_{\text{bound}} \;\geq\; q_{\min}$, no unvisited entries in S_i can improve $q_{\min}$, and the scan is stopped. After this bounded scan of all sets, it is guaranteed that the correct $q_{\min}$ has been found. This is the approach of `RapidNJ`.

Improving the d-filter While this method is correct, and provides dramatic speed gains [14], it can be improved. First, observe that the bound is dependent on $t_{\max}$, which may be very loose (see fig. 2a). One way to provide tighter bounds is to abandon the idea of creating one list per cluster. Instead, the interval ($t_{\min}$, $t_{\max}$) is divided into evenly spaced disjoint bins, where each bin B_x covers the interval $[T_x^{\min}, T_x^{\max})$. For X bins, then, the size of the interval between min and max values will be $(t_{\max} - t_{\min})/X$ (the default number of bins is 30). Each cluster i is associated with the bin B_x for which $T_x^{\min} \;\leq\; t_i \;<\; T_x^{\max}$. Adopt the notation that cluster i's bin $B(i) = x$. Note that bins may contain differing numbers of clusters. Then create a set $S_{\{x,y\}}$ for each bin-pair $\{B_x, B_y\}$.

Now, instead of placing $(\{i, j\}, d_{ij})$ triples into per-cluster sets as before, place them in per-bin-pair sets $S_{\{B(i),B(j)\}}$, still sorting triples within a set by increasing d_{ij}. To find $q_{\min}$, traverse the sets, scanning through each as before, but now calculating the bound based on current triple $(\{i, j\}, d_{ij})$, taken from set $S_{\{x,y\}}$, as

$$q_{\text{bound}} = (r-2)\; d_{ij} - T_x^{\max} - T_y^{\max}. \tag{4}$$

This improves the filter because, for an unvisited pair $\{i', j'\}$ from the same set $S_{\{x,y\}}$, setting $\rho = (r-2)\; d_{i'j'}$, $\rho - T_x^{\max} - T_y^{\max}$ will usually be a tighter bound on $q_{i'j'}$ than is $\rho - t_{i'} - t_{\max}$.

Updating data structures After merging clusters i and j, the rows and columns associated with those columns are inactivated in D, and a new row and column are added for the merged cluster ij. Entries in the sets also require update. The new cluster, ij, is associated with the bin $B(ij) = \text{argmin}_x\{T_x^{\max} > t_{ij})\}$. Triples $(\{ij, k\}, d_{ij|k})$ for distances to each remaining cluster are added to the appropriate set, $S_{\{B(ij),B(k)\}}$. Triples for the removed clusters i and j are removed from sets in a lazy fashion: i and j are marked as retired, and when triples involving either i or j are encountered while scanning sets, they are removed.

While this method provides tighter q_{bound} values than the method of keeping one set per cluster, these bounds will tend to relatively loosen over time. Before any merges are performed, the intervals of these sets are non-overlapping, but because the change in t_k after a merge may be different for each cluster k, this non-overlapping property is no longer guaranteed to hold after a merge is performed. The result is a loosening of the value $T_x^{\max}$ as a bound for t_i for an arbitrary cluster (i.e. the bound may be greater than $(t_{\max} - t_{\min})/X$). The loosening of the bound grows as iterations pass, though it is still tighter than the per-cluster bound until the set ranges overlap almost completely.

It may seem appealing to move a cluster to a new bin when that bin could provide a tighter bound, but doing so would incur substantial work to take all corresponding triples out of the various bin-pair sets. The strategy taken by `NINJA` is to occasionally rebuild the sets from scratch. For a constant $K > 1$ (the default is $K = 2$), the sets are rebuilt after r/K merges have been performed since the last rebuild, where r is the number of clusters remaining at the time of that prior rebuild. Overall runtime for these set constructions is dominated by the time of the first construction, $O(n^2 \log n)$.

3.2 Overcoming memory limits

The size of D is quadratic in the number of sequences, as is the size of the sets of triples described above. If these structures grow to exceed available RAM, an application may either abort or store the structures to *external storage* (i.e. disk). If the pattern of disk access is is random, the latter will result in frequent paging. The dramatic difference in latency between disk and RAM access (on the order of 10^6-fold difference [21]) necessitates I/O-efficient algorithms if external storage is to be used. I describe methods for efficiently handling both the sets $S_{\{x,y\}}$ and the distance matrix.

Bin-pair sets in external memory The set of triples associated with each bin pair set $S_{\{x,y\}}$ has been described as a sorted list. In fact, in order to allow fast insertion of triples for new clusters, such a list would likely be implemented as a data structure such as a binary search tree. Binary search trees have poor I/O behavior when stored to disk, but could be easily replaced by a B-tree [22] or B+ tree, which allow for logarithmic number of disk I/Os for both insertions and reads.

However, since only a small portion of the entries in a set are accessed, the effort of keeping a totally ordered data structure is unnecessary. A min-heap [23] provides the tools necessary to scan through increasing d_{ij}, with less overhead since it only need keep a partial order. `NINJA` implements an *external memory array heap* [24], keyed on d_{ij}. This heap structure can store more triples than would fill a 1TB hard drive while maintaining a memory footprint smaller than 2MB, and guarantees an ammortized number of I/O operations for *insert* and *extract-min* operations that is logarthmic in the number of inserted triples. One heap is used for each set $S_{\{x,y\}}$.

Distance matrix in external memory Though the heaps are used to identify the cluster pair $\{i, j\}$ to merge, the distance matrix D should still be maintained. After a merge, $d_{ij|k}$ is

calculated for every cluster k. From equation 3, we see that we must view d_{ik} and d_{jk} for every k, which is more efficiently done by traversing the rows and columns for i and j in D than by scanning through the heaps.

Since `NJ` expects D to be symmetric, an efficient way to store D for in-memory use is to keep only its upper-right triangle: distances for cluster i are spread across row i and column i, such that all reside in the upper triangle. When a pair of clusters $\{i, j\}$ is merged, a new row and column are said to be added, but no additional space is actually required: the distances of the new cluster ij to all remaining clusters k can be stored in the cells previously belonging to one of the retired clusters, say i, so $d_{ij|k}$ is stored in the cell where d_{ik} was stored. Clusters i and j are noted as retired, and the mapping of cluster ij's stored location is simple.

However, when D is stored to disk, this approach will lead to poor disk paging behavior, because values for cluster i are split between row i (which can be accessed efficiently from disk, with many consecutive values per disk block), and column i (which will be spread across the disk, with typically one value per disk block). Therefore, a modification is required. For an input of n sequences, a file F stores a matrix with with $2n$ columns and n rows. The full initial D (i.e. both the upper and lower triangles) is stored to F, filling the first n columns for each row. When a merge is performed, and new distances are calculated, the values d_{ik} and d_{jk} can be gathered by sweeping through rows i and j, allowing the number of distance values that fit in a disk page to be gathered at the cost of a single disk access. The mapping for the storage location of the new $d_{ij|k}$ values will be different for rows and columns: if ij is formed as the result of the pth merge, then it will map to the row in F where i was stored, but will fill a new column $n + p - 1$. Newly calculated distances are not immediately stored to disk, instead waiting until enough values have been calculated to allow for efficient disk I/O. Suppose b distance values fit in a disk block: then d_{ij}s for new clusters are appended to a b x n in-memory matrix M until all b columns of that matrix are full. At that time, each row of M is appended to the same row in F (requiring one disk I/O per row), and each column is translated and written into the mapped row in F (requiring up to $\lceil n/b \rceil$ I/Os).

4 Candidate handling

Due to the nature of heaps, all viewed $(\{i, j\}, d_{ij})$ triples are removed from their containing heaps during the search for $q_{\min}$; call these the *candidates*. The d-filter method described in section 3 dramatically reduces the number of candidates viewed in most cases, but inputs with relationships like those seen in figure 2a reduce the efficacy of d-filtering, for reasons described in section 6. Examples of the impact on run time are given in table 2c.

Here I describe a second level of filtering, called the *q-filter*. It works by sequestering candidates passing the d-filter, and organizing them in a way that allows a new bound to limit the number of those candidates that are viewed in each iteration.

q-filter on a candidate heap Let $q_{ij}(p)$, $r(p)$, and $t_i(p)$ correspond to the values of q_{ij}, r, and t_i at a fixed previous iteration p. And let $\delta_i(p) = (r-2)t_i(p) - (r(p)-2)t_i$. Then it is easy to show that, for the current iteration,

$$q_{ij} \;=\; \frac{(r-2)\,q_{ij}(p) \;+\delta_i(p) + \delta_j(p)}{r(p)-2}. \tag{5}$$

Suppose all candidates on hand at iteration p are stored as $(\{i, j\}, q_{ij}(p))$ triples in a candidate set, sorted according to their $q_{ij}(p)$ values. Assign the current r and t_i as $r(p)$ and $t_i(p)$ for that set. Since relative q-values change by small amounts from one iteration to the next, the $\{i, j\}$ pair with the smallest q_{ij} at a future iteration is likely to be near the front of

this sorted list. It can be found by initializing $q_{\min}$ to ∞, then scanning candidates in order of increaing $q_{ij}(p)$, updating $q_{\min}$ when an entry with a smaller q_{ij} is found.

Let \$ be the set of all clusters with at least one representative in the candidate set, and define

$$\Delta_{\max}(p) := \max_{\substack{i,j \in \$ \\ i \neq j}} \{\delta_i(p) + \delta_j(p)\} \ . \tag{6}$$

Then scanning of this sorted list may be stopped when an element is found with

$$\frac{(r-2)\, q_{ij}(p) \ + \Delta_{\max}(p)}{r(p) - 2} \ \geq \ q_{\min} \ . \tag{7}$$

The candidate set can be large enough to exceed memory for very large inputs, and because only a partial order is required, `NINJA` stores the contents of the candidate set in an external memory heap array, as described for the d-bound bin-pairs in section 3. The heap formed from such candidates is called a *candidate heap*.

Candidate heap chain Adding a new candidate to a candidate heap created in a previous iteration p_a (with associated $r(p_a)$ and $t(p_a)$ values) is problematic: (1) if the candidate involves a cluster j that was formed after p_a, then $q_{ij}(p_a)$ and $t_j(p_a)$ are undefined, and (2) even if both clusters existed before p_a, the candidate would need to be stored on the heap with a back-calculated $q_{ij}(p_a)$ (and thus looser than necessary bounds) to retain sensible δ-values.

The response in `NINJA` is to keep a chain of candidate heaps. At initiation, there are no candidates. In each iteration, newly gathered candidates from the d-filter are placed in a single candidate pool. When the size of that pool exceeds a threshold (default is 50,000; it should be fairly large because of the overhead required to form an external memory array heap), a candidate heap is created and populated with the triples in the pool, and the pool is then emptied. As more candidates are gathered, they are again stored in the pool, until it exceeds threshold, at which time a second candidate heap is formed, filled from the candidate pool, and linked to the first. This is repeated until the tree is complete. This results in a chain of candidate heaps. The chain is destroyed when bin-pair heaps are rebuilt (section 3.1).

At each iteration, these heaps are scanned for elements with small q_{ij} by removing triples until the bound (7) is reached. Those viewed triples with $q_{ij} > q_{\min}$ are placed in the candidate pool, rather than being returned to their source candidate heap, because the δ-bound usually gets looser, so they'd almost always just be pulled back off their original heap on the next iteration. When a candidate heap drops below a certain size (default = 60% of original size), it is liquidated, and all triples placed in the candidate pool.

5 Algorithm overview

At each iteration p_a, `NINJA` follows this process, tracking $q_{\min}$ at each step:

1. Scan all candidates in the pool, keeping the one with smallest q_{ij}.
2. Sweep through the candidate heap chain, for each heap removing triples until reaching the bound (7), and placing those triples in the candidate pool. Possibly liquidate heaps in the chain if they become too empty. Steps 1 and 2 typically provide a good bound on the best q_{ij} value for the iteration, because they start with a set of previously filtered candidates.
3. Sweep through the bin-pair heaps, for each heap removing triples until reaching bound (4), and placing those triples in the candidate pool.
4. If the size of the candidate pool exceeds threshold, move all candidates into a new heap, storing $q_{ij}(p_a)$ for each candidate, and $t_i(p_a)$-values and $r(p_a)$ for the heap. Append this heap to the candidate heap chain.

5. Having found the q_{ij} with minimum value, merge clusters i and j, update the bin-pair heaps and the in-memory part of the distance matrix M with entries for new cluster ij, and possibly write out to the on-disk distance matrix D. Also occasionally liquidate the candidate heap chain and rebuild the bin-pair heaps (see section 3.1).

6 Results and discussion

To assess the effectiveness of the two-tiered filtering algorithm, I have implemented it in an application called `NINJA`. Three variants were used in various tests in the results shown below. The default variant, `NINJA`, stores the distance matrix on disk, and uses both the d-filter described in section 3 and the q-filter described in section 4, both implemented with external-memory array heaps [24]. The variant labeled `NINJA-d-filter` is identical to `NINJA`, except that it implements only the d-filter, not the q-filter. The variant labeled `NINJA-InMem` also uses only the d-filter, but does so with in-memory data structures - keeping the distance matrix entirely in memory, and using a binary heap in place of the external-memory array heap. `NINJA-InMem` makes it possible to directly assess the impact of external-memory components of the algorithm. On a machine with 4GB RAM, it is only able to compute neighbor-joining trees on inputs of fewer than about 7000 sequences, due to overhead memory use.

For comparison purposes, I tested two tools that similarly avoid viewing the entire distance matrix at each iteration, `QuickJoin` and `RapidNJ`, and a very fast implementation of the canonical algorithm, `QuickTree`. To my knowledge, these are the fastest available tools that implement exact `NJ`. Both of the former tools are unable to handle inputs of more than 13,000 sequences on a machine with 4GB of RAM, but an experimental external-memory version of `RapidNJ`, called `RapidDiskNJ`, has been released. A verison of `RapidDiskNJ` downloaded on 04/24/09 was used as a reference for large inputs. Note that the filter used in `RapidNJ` and `RapidDiskNJ` is almost equivalent to that used in `NINJA-d-filter` and `NINJA-InMem`. Tree constructing methods that do not form `NJ` trees are not included in this analysis due to space limits. It is worth noting that `ClearCut` and `FastTree` are both faster than `NINJA`.

`QuickTree` was implemented in *C*, `QuickJoin` and both `RapidNJ` variants were implemented in *C++*, and the `NINJA` variants were implemented in *Java*.

Environment Experiments were run on a bank of 8 identical dedicated systems running CentOS 4.5 (kernel 2.6.9-55), with 64 bit 2.33 GHz Xeon processors, 4 GB allocated RAM, and 500 GB 7200 RPM SATA hard drives. `NINJA` used roughly 60 GB of disk space for the largest inputs. The "real time" output from the standard `time` tool was used to measure run time.

Data `Pfam` [7] families were used as sample input for the tools. Each protein domain family was preprocessed to remove duplicate sequences, and all 415 families with more than 2000 unique sequences were used. Phylip formatted distance matrices, calculated with `QuickTree`, were used as input to all tools.

Effect of filters At each iteration, the canonical algorithm scans through all $r(r-1)/2$ cells in the distance matrix, where r is the number of remaining clusters. It thus views $\Theta(n^3)$ cells (candidates) over the course of building a complete `NJ` tree on n sequences. Figure 1 shows the often dramatic reduction in number of candidates passing the d-filter, relative to this total count of cells. It also highlights instances where the d-filter is mostly ineffective, and shows that more consistent success is achieved when the q-filter is used in conjuction with the d-filter. It is important to note that figures 1, 3, and 4 are all log-log plots. Thus, the roughly linear growth

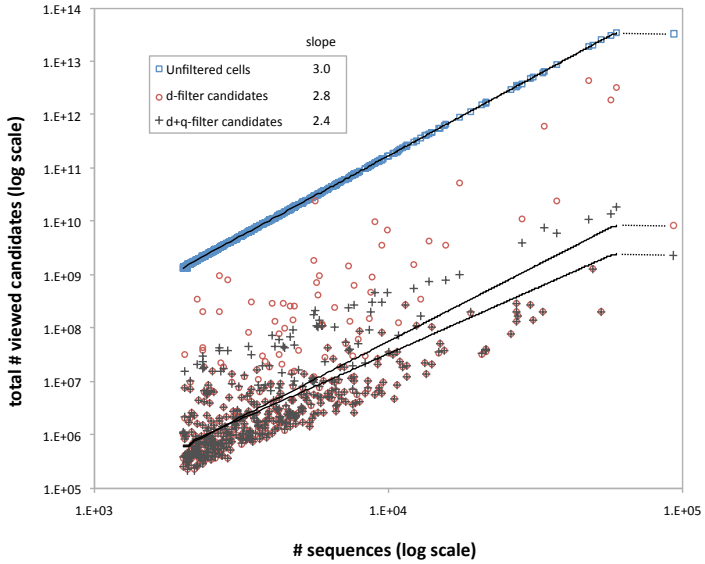


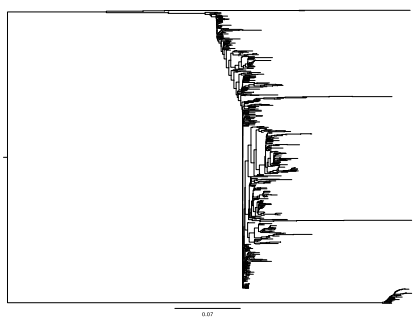
Figure 1 Number of candidates viewed during tree-building with and without filters, for all 415 Pfam alignments with more than 2000 non-duplicate sequences. Data points are placed on a log-log plot: the slope on such a plot gives the exponent of growth. The canonical algorithm treats all cells as candidates, and the coresponding count of unfiltered cells shows the expected slope of 3 for $\Theta(n^3)$ number of cells. The d-filter often reduces the number of viewed candidates by more than 3 orders of magnitude, but is less effective for some inputs. Addition of the q-filter results in more consistent filtering success across all inputs, and an observed growth rate in number of viewed cells of roughly $O(n^{2.4})$.

observed in all plots corresponds to polynomial growth of both candidates and run time, with the polynomial exponent visible in the log-log slope.

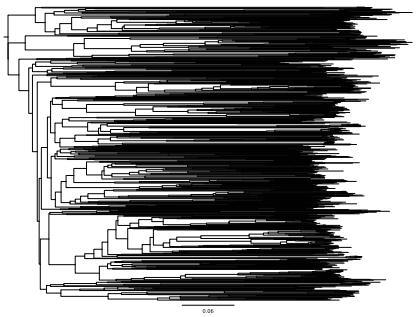
Inputs causing bad d-filtration Figure 2a shows an example of the kind of input that makes the d-filter fairly ineffective. It contains a large clusters of very closely related sequences, and a few relatively long branches. Contrast this to the more evenly-distributed sequences seen in figure 2b, for which the d-filter is quite effective.

The reason for the computational difficulty of trees like the one in figure 2a is that the clusters on the very long branches have very large t values relative to the t values for most clusters, while the clusters for the tree in figure 2b will all have fairly similar t values. With `RapidNJ`'s bound, which depends on $t_{\max}$, d-filtering on 2a is immediately inefficient because of this t-value discrepancy. `NINJA`'s bound (equation 4) starts off relatively tight, but the d-filtering becomes inefficient as the range of t-values within a bin grows. This can happen dramatically when clusters along one long branch, which thus begin in a high t-value bin, are merged (with corresponding relative reduction in t values), while other clusters sharing the same bin are not merged, and thus retain relatively high t values

Table 2c shows the effect that these differing tree forms have on both number of viewed candidates and run time. Focus on the results for family Cytochrom_B_N, an input with structure like that shown in figure 2a: d-filtering only reduces the number of candidates by a factor of 10, much less effective filtering than the 10,000-fold reduction seen in family WD40, an input with structure much like that seen in figure 2b. Because of the extra overhead of their



(a) Flu_M1, 707 sequences. Example of a topology for which filtering is ineffective.



(b) QRPTase_N, 707 sequences. A topology for which filtering is effective.

		Number candidates			Run time (minutes)			
Pfam ID	Sequence number	All cells	d-filter only	d+q filters	QuickTree	RapidDiskNJ	NINJA d-filter	NINJA d+q filters
RuBisCO_large	17,490	9E+11	5E+10	1E+09	64	124	331	25
PPR	18,961	1E+12	2E+08	2E+08	155	20	21	20
Cytochrom_B_N	33,789	6E+12	6E+11	7E+09	539	1,678	6,092	146
WD40	33,327	6E+12	2E+08	2E+08	756	52	121	110
RVT_1	56,822	3E+13	2E+12	1E+10	n/a	>18,500	>18,500	717
ABC_tran	53,116	2E+13	2E+08	2E+08	n/a	159	554	530

(c) Impact of d- and q-filtering at various input sizes.

Figure 2 Trees (a) and (b) are both of 707 sequences, and represent approximately equal evolutionary distance between the most divergent pair of sequences. They are mid-point-rooted, and images were created using `FigTree` (http://tree.bio.ed.ac.uk/software/figtree/). `Pfam` datasets with relationships like those shown in (a), with many closely related sequences and a few relatively long branches, cause d-filtering to be ineffective. In table (c), the first of each pair has topology similar to (a), and shows poor d-filtering; the second has topology similar to (b), and shows good d-filtering. Run times for `RapidNJ` and `NINJA-d-filter` are very slow when d-filtering is ineffective, but the additional q-filter used by `NINJA` results in much better filtering, and therefore improves runtimes even for these hard cases. On a system with 4GB RAM, `QuickTree` crashes on all `Pfam` families with more than 37,000 sequences. `RapidDiskNJ` and `NINJA-d-filter` both took longer than 13 days to compute a tree for RVT_1.

algorithms, the resulting run times for both `RapidDiskNJ` and `NINJA-d-filter` are much worse than that of `QuickTree`. By applying the q-filter, `NINJA` achieves a further 100-fold reduction in candidates viewed for Cytochrom_B_N, along with a large reduction in run time.

Comparison to other tools Figures 3 and 4 compare `NINJA` variants to other neighbor-joining tools. They focus on inputs of more than 2000 sequences, since smaller inputs are solved by the canonical algorithm (implemented in `QuickTree`) in under 10 seconds.

The orders-of-magnitude reduction in viewed candidates seen in figure 1 does not translate to a similar reduction in run-time because the underlying data structures required to gain this filtering advantage incur a great deal of overhead relative to the simple scanning of a matrix. In addition, the large-scale applications (`NINJA` and `RapidDiskNJ`) incur a constant-factor overhead from disk accesses. Those factors are mitigated by using algorithms with good disk-paging behavior, but are nevertheless present.

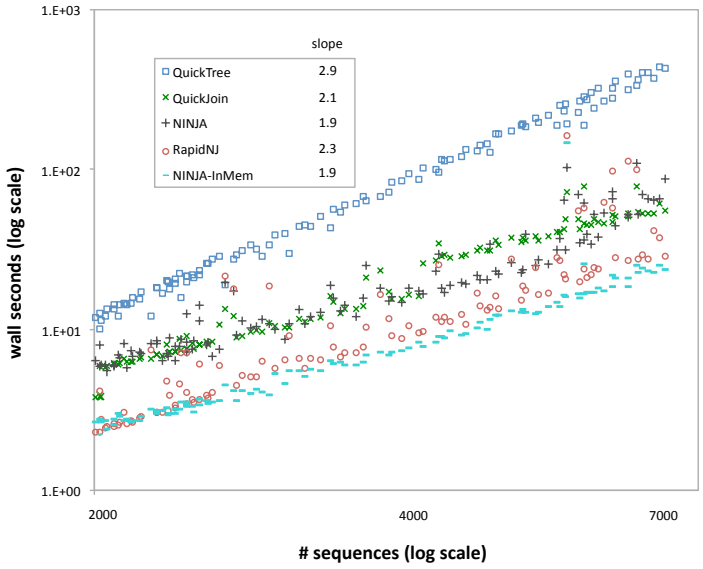


Figure 3 Performance of `NINJA` and `NINJA-InMem` compared to that of `QuickTree`, `QuickJoin`, and `RapidNJ` on a random sample of medium-sized (2000 to 7000 sequences) `Pfam` inputs.

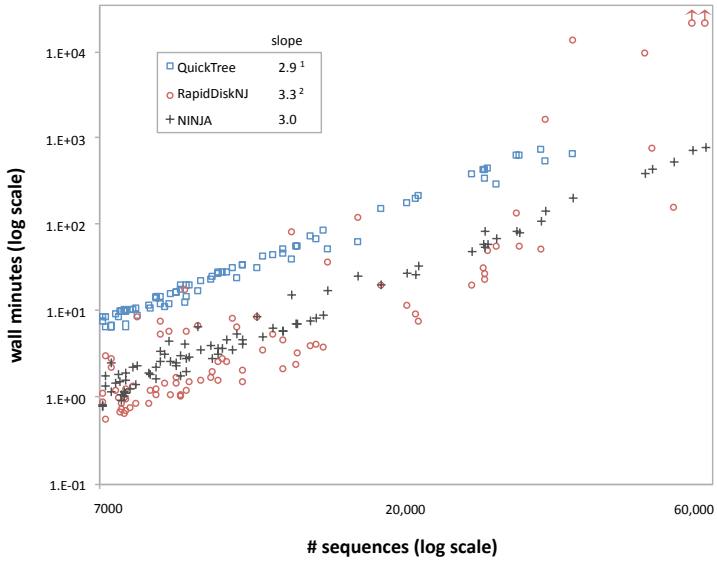


Figure 4 Performance of `NINJA` compared to that of `QuickTree` and `RapidDiskNJ` on all large (7000 to 60,000 sequences) `Pfam` inputs. (1) On a system with 4GB RAM, `QuickTree` crashes on inputs with more than 37,000 sequences. (2) `RapidDiskNJ` failed to complete within 13 days for the two largest inputs; the uncertain times-to-completion are represented with the arrowed cirles in the upper right corner. The slope for `RapidDiskNJ`, which shows that its run time is growing faster than n^3, does not include these two points.

Figure 3 shows run times for a random sample of medium-sized (2000-7000 sequences) inputs from Pfam. A sample is shown, rather than the entire datset, to improve visibility of the chart, and agrees with trends for the full set of similarly-sized inputs. Note that QuickTree's run time grows with a slope of 2.9 on a log-log plot, essentially what is expected of a $\Theta(n^3)$ algorithm. QuickJoin and RapidNJ are in-memory versions of competitor algorithms - both show a reduction in run-time, and a growth rate that is slightly more than quadratic. This is in agreement with results from [14]. Results for NINJA-InMem and NINJA are presented to show their relative performance to each other and the other tools. Both show a roughly quadratic run-time growth on this data set. NINJA-InMem is slightly faster than the fastest other tool, RapidNJ. Since the two tools use essentially the same bounding method for their d-filter methods, this difference is likely explained by the tighter bounds generated by the bin-pair approach of NINJA.

Figure 4 shows run times for all inputs from Pfam with more than 7000 sequences. Results are given for the variant of each tool that best handles these large inputs: QuickTree, RapidDiskNJ, and NINJA. Only NINJA successfully computed NJ trees for all inputs; QuickTree crashed on all inputs with more than 37,000 sequences, while RapidDiskNJ failed to complete within 13 days on the two largest inputs. QuickTree continues to exhibit the expected slope (3.0) on a log-log plot for a $O(n^3)$ algorithm. Interestingly, both RapidDiskNJ and NINJA also show a similar cubic slope for these larger inputs, in conflict with the lower rate of growth observed for smaller inputs in figure 3 and [14]. Inspection of the data suggests that this is due to an increased frequency in these larger datasets of the sort of difficult inputs characterized by figure 2a. Note that the number of viewed candidates was observed in figure 1 as growing with a power of 2.4. The logarithmic overhead of heap data structures is responsible for the observation that run time grows faster than the number of candidates.

7 Conclusion

I have presented a new tool, NINJA, that builds a tree under the traditional optimization criteria of NJ, with the associated guarantee of statistical consistency. NINJA speeds up NJ by employing a two-tiered filtering regime, which greatly reduces the number of viewed candidates in each iteration relative to the complete scan of the distance matrix that is employed in the canonical algorithm. NINJA also overcomes memory constraints seen in earlier filtering-based work by incorporating external-memory-efficient data structures into the algorithm, specifically the external memory array heap [24] and simple on-disk storage of the distance matrix. The latter structure can be trivially co-opted by any NJ tool to overcome memory constraints due to the size of the distance matrix.

Though this method greatly speeds up NJ, and makes it possible to construct extremely large NJ trees, the run time still appears to be in $O(n^3)$ despite the dramatic reduction in viewed candidates. Though this does not represent an improvement in growth rate, the reduced constant factor makes it feasible to construct trees for inputs with well over 100,000 sequences in a matter of a small number of days of computation on a modern desktop.

The accuracy of NINJA is not discussed in this paper, as accuracy of any exact NJ tool is expected to be the same. That said, it is a straightforward exercise to incorporate the variance-minimization calculations of BioNJ [25], which have been show to improve accuracy over canonical neighbor-joining, into NINJAś algorithm.

Acknowledgements I thank Karen Cranston, John Kececioglu, Morgan Price, and Mike Sanderson for helpful discussions, and Mike Sanderson and Darren Boss for use of, and assistance with, Mike's computing cluster.

I am supported by a PhD Fellowship from the University of Arizona National Science Foundation IGERT Comparative Genomics Initiative Grant DGE-0654435.

References

1. Saitou, N., Nei, M.: The neighbor-joining method: a new method for reconstructing phylogenetic trees. Mol. Biol. Evol **4** (1987) 406–425
2. Nakhleh, L., Moret, B.M.E., Roshan, U., John, K.S., Sun, J., Warnow, T.: The accuracy of fast phylogenetic methods for large datasets. In Proc. 7th Pacific Symp. on Biocomputing PSB 02 (2002) 211–222
3. Atteson, K.: The Performance of Neighbor-Joining Methods of Phylogenetic Reconstruction. Algorithmica **25** (1999) 251–278
4. Felsenstein, J.: Inferring phylogenies. (Jan 2004)
5. Bryant, D.: On the Uniqueness of the Selection Criterion in Neighbor-Joining. Journal of Classification **22** (2005) 3–15
6. Studier, J.A., Keppler, K.J.: A note on the neighbor-joining algorithm of Saitou and Nei. Mol Biol Evol **5**(6) (11 1988) 729–31
7. Finn, R.D., Tate, J., Mistry, J., Coggill, P.C., Sammut, S.J., Hotz, H.R.R., Ceric, G., Forslund, K., Eddy, S.R., Sonnhammer, E.L.L., Bateman, A.: The Pfam protein families database. Nucleic Acids Res **36**(Database issue) (1 2008) D281–8
8. Griffiths Jones, S., Moxon, S., Marshall, M., Khanna, A., Eddy, S.R., Bateman, A.: Rfam: annotating non-coding RNAs in complete genomes. Nucleic Acids Res **33**(Database issue) (1 2005) D121–4
9. Goldman, N., Yang, Z.: Introduction. Statistical and computational challenges in molecular phylogenetics and evolution. Philos Trans R Soc Lond B Biol Sci **363**(1512) (12 2008) 3889–92
10. Smith, S.A., Beaulieu, J.M., Donoghue, M.J.: Mega-phylogeny approach for comparative biology: an alternative to supertree and supermatrix approaches. BMC Evol Biol **9** (2009) 37
11. Howe, K., Bateman, A., Durbin, R.: QuickTree: building huge Neighbour-Joining trees of protein sequences. Bioinformatics **18**(11) (11 2002) 1546–7
12. Mailund, T., Pedersen, C.N.S.: QuickJoin–fast neighbour-joining tree reconstruction. Bioinformatics **20**(17) (11 2004) 3261–2
13. Mailund, T., Brodal, G.S., Fagerberg, R., Pedersen, C.N.S., Phillips, D.: Recrafting the neighbor-joining method. BMC Bioinformatics **7** (2006) 29
14. Simonsen, M., Mailund, T., Pedersen, C.N.S.: Rapid Neighbor-Joining. WABI '08: Proceedings of the 8th international workshop on Algorithms in Bioinformatics (2008) 113–122
15. Zaslavsky, L., Tatusova, T.: Accelerating the neighbor-joining algorithm using the adaptive bucket data structure. Lecture notes in computer science **4983** (2008) 122
16. Evans, J., Sheneman, L., Foster, J.: Relaxed neighbor joining: a fast distance-based phylogenetic tree construction method. J Mol Evol **62**(6) (6 2006) 785–92
17. Elias, I., Lagergren, J.: Fast Neighbor Joining. Theor. Comput. Sci. **410** (2009) 1993–2000
18. Sheneman, L., Evans, J., Foster, J.A.: Clearcut: a fast implementation of relaxed neighbor joining. Bioinformatics **22**(22) (11 2006) 2823–4
19. Desper, R., Gascuel, O.: Fast and accurate phylogeny reconstruction algorithms based on the minimum-evolution principle. Journal of computational biology **9**(5) (2002) 687–705
20. Price, M.N., Dehal, P.S., Arkin, A.P.: FastTree: Computing Large Minimum-Evolution Trees with Profiles instead of a Distance Matrix. Mol Biol Evol (2009) to appear, doi:10.1093/molbev/msp077
21. Patterson, D.A.: Latency lags bandwidth. Communications of the ACM **47**(10) (10 2004) 71–75
22. Bayer, R., McCreight, E.: Organization and Maintenance of Large Ordered Indexes. Acta Informatica **1** (1972) 173–189
23. Corman, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.: Introduction to algorithms. 2 edn. MIT press, Cambridge, MA, USA (2001)
24. Brengel, K., Crauser, A., Ferragina, P., Meyer, U.: An Experimental Study of Priority Queues in External Memory. WAE '99: Proceedings of the 3rd International Workshop on Algorithm Engineering (1999) 345–359
25. Gascuel, O.: BIONJ: an improved version of the NJ algorithm based on a simple model of sequence data. Mol Biol Evol **14**(7) (7 1997) 685–95